CSC 405 Control Hijacking Attacks, Part Deux (Defenses)

Adam Gaweda
agaweda@ncsu.edu

Alexandros Kapravelos
akaprav@ncsu.edu

How can we prevent a buffer overflow?

StackGuard

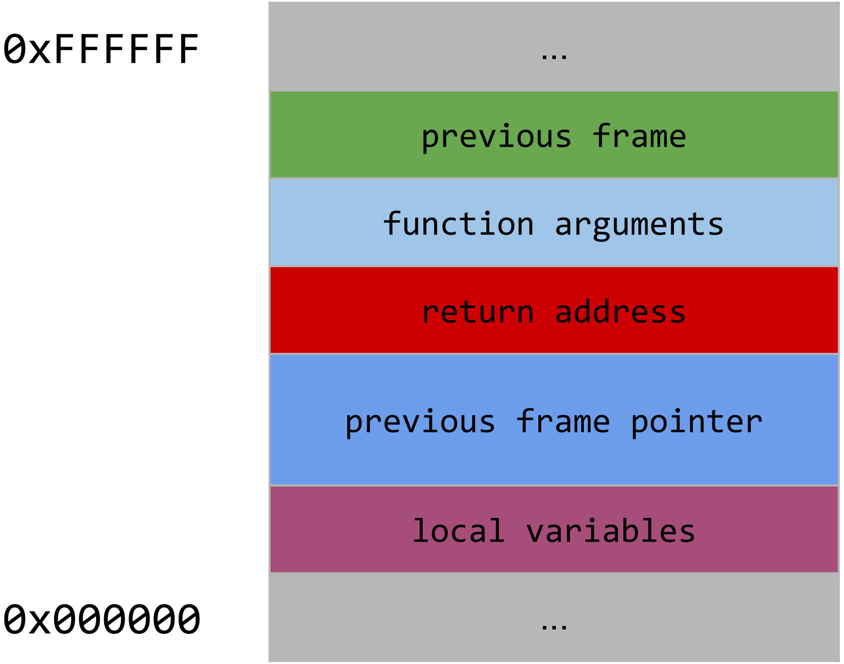
- A compiler technique that attempts to eliminate buffer overflow vulnerabilities

- No source code changes

- Patch for the function **prologue** and **epilogue**

- **Prologue**
  - push an additional value into the stack (**canary**)
- **Epilogue**
  - pop the canary value from the stack and check that it hasn’t changed

Regular Stack



0xFFFFFF

...

previous frame

function arguments

return address

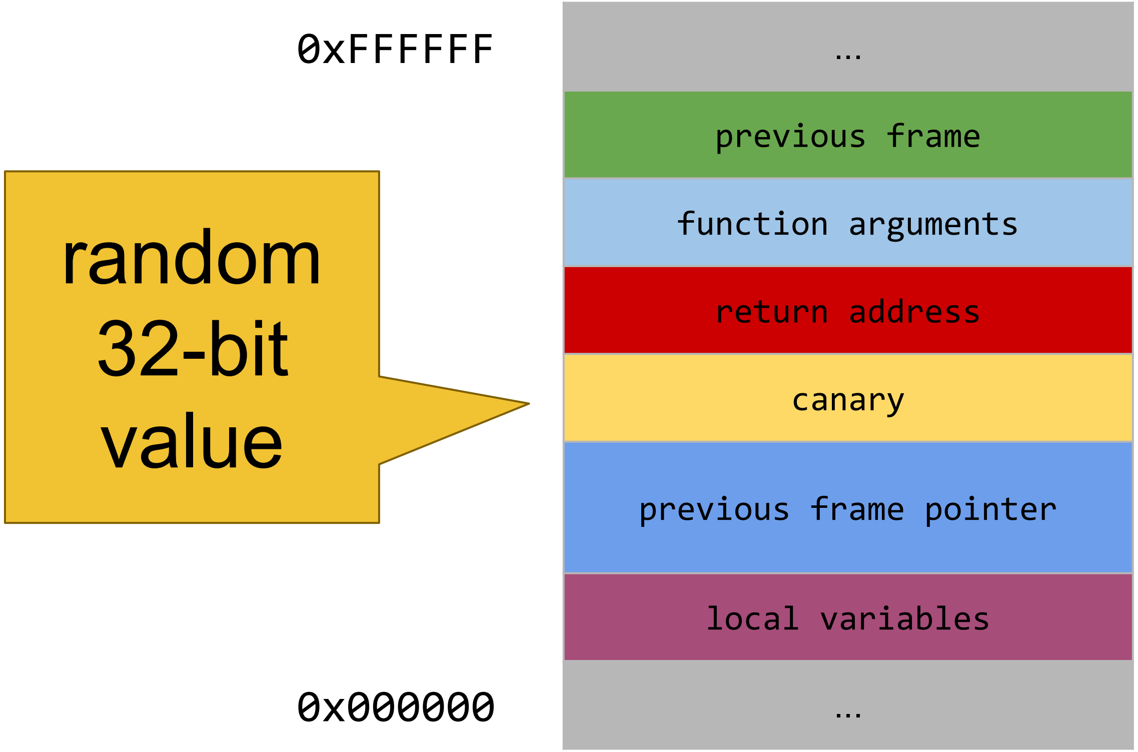
previous frame pointer

local variables

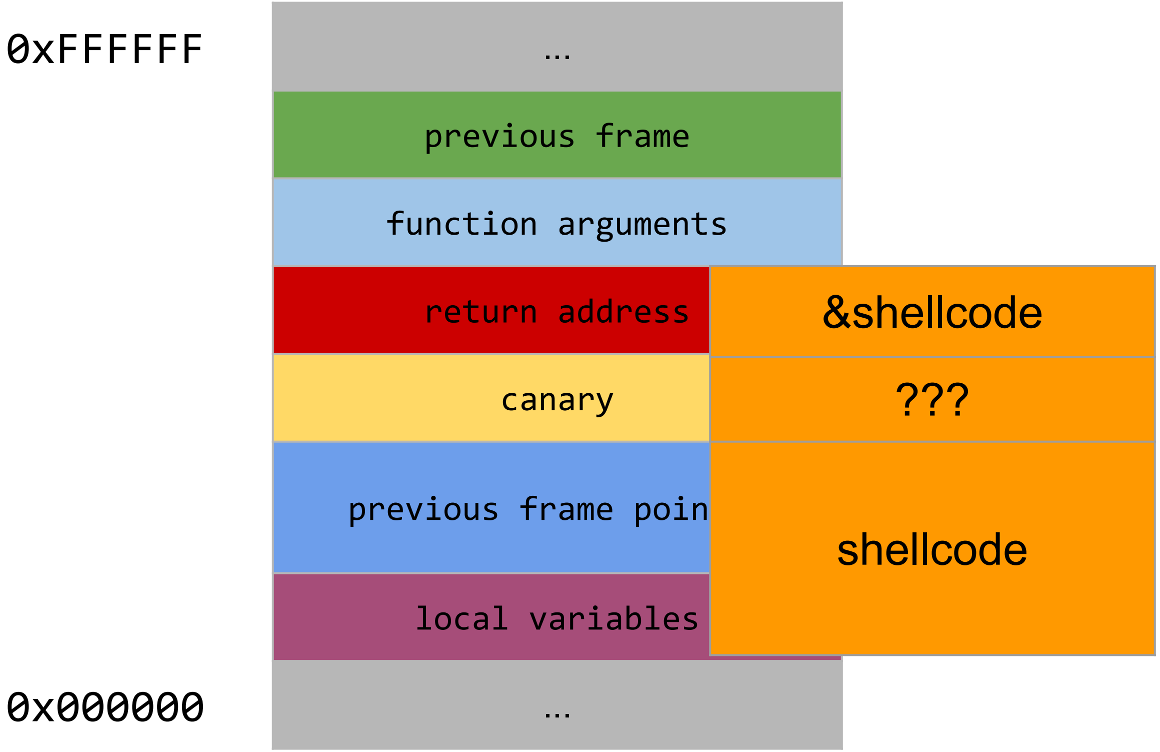
0x000000

...

Stack Guarded by Canary



StackGuard


0xFFFFFF
...
previous frame
function arguments
return address
canary
previous frame poin
local variables
...
0x000000
&shellcode
???
shellcode

Let’s check what gcc does!

```
#include <stdio.h>

int main(void) {
  return printf("Hello World!\n");
}



$ gcc -fstack-protector-all helloworld.c -o helloworld
$ gdb ./helloworld
```

StackGuard Assembly - Prologue

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804846b <+0>:   lea    0x4(%esp),%ecx
   0x0804846f <+4>:   and    $0xfffffff0,%esp
   0x08048472 <+7>:   pushl  -0x4(%ecx)
   0x08048475 <+10>:  push   %ebp
   0x08048476 <+11>:  mov    %esp,%ebp
   0x08048478 <+13>:  push   %ecx
   0x08048479 <+14>:  sub    $0x14,%esp
   0x0804847c <+17>:  mov    %gs:0x14,%eax
   0x08048482 <+23>:  mov    %eax,-0xc(%ebp)
   0x08048485 <+26>:  xor    %eax,%eax
   0x08048487 <+28>:  sub    $0xc,%esp
   0x0804848a <+31>:  push   $0x8048530
   0x0804848f <+36>:  call   0x8048330 <printf@plt>
```

StackGuard Assembly - Prologue

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804846b <+0>:   lea    0x4(%esp),%ecx
   0x0804846f <+4>:   and    $0xfffffff0,%esp
   0x08048472 <+7>:   pushl  -0x4(%ecx)
   0x08048475 <+10>:  push   %ebp
   0x08048476 <+11>:  mov    %esp,%ebp
   0x08048478 <+13>:  push   %ecx
   0x08048479 <+14>:  sub    $0x14,%esp
   0x0804847c <+17>:  mov    %gs:0x14,%eax
   0x08048482 <+23>:  mov    %eax,-0xc(%ebp)
   0x08048485 <+26>:  xor    %eax,%eax
   0x08048487 <+28>:  sub    $0xc,%esp
   0x0804848a <+31>:  push   $0x8048530
   0x0804848f <+36>:  call   0x8048330 <printf@plt>
```

Generate canary and wipe the evidence

StackGuard Assembly - Epilogue

```
   0x08048494 <+41>:  add    $0x10,%esp
   0x08048497 <+44>:  mov    -0xc(%ebp),%edx
   0x0804849a <+47>:  xor    %gs:0x14,%edx
   0x080484a1 <+54>:  je     0x80484a8 <main+61>
   0x080484a3 <+56>:  call   0x8048340 <__stack_chk_fail@plt>
   0x080484a8 <+61>:  mov    -0x4(%ebp),%ecx
   0x080484ab <+64>:  leave
   0x080484ac <+65>:  lea    -0x4(%ecx),%esp
   0x080484af <+68>:  ret
End of assembler dump.
```

StackGuard Assembly - Epilogue

```
   0x08048494 <+41>:   add    $0x10,%esp
   0x08048497 <+44>:   mov    -0xc(%ebp),%edx
   0x0804849a <+47>:   xor    %gs:0x14,%edx
   0x080484a1 <+54>:   je     0x80484a8 <main+61>
   0x080484a3 <+56>:   call   0x8048340 <__stack_chk_fail@plt>
   0x080484a8 <+61>:   mov    -0x4(%ebp),%ecx
   0x080484ab <+64>:   leave
   0x080484ac <+65>:   lea    -0x4(%ecx),%esp
   0x080484af <+68>:   ret
End of assembler dump.
```

Check if the canary is "still alive" (unchanged)

Canary Types

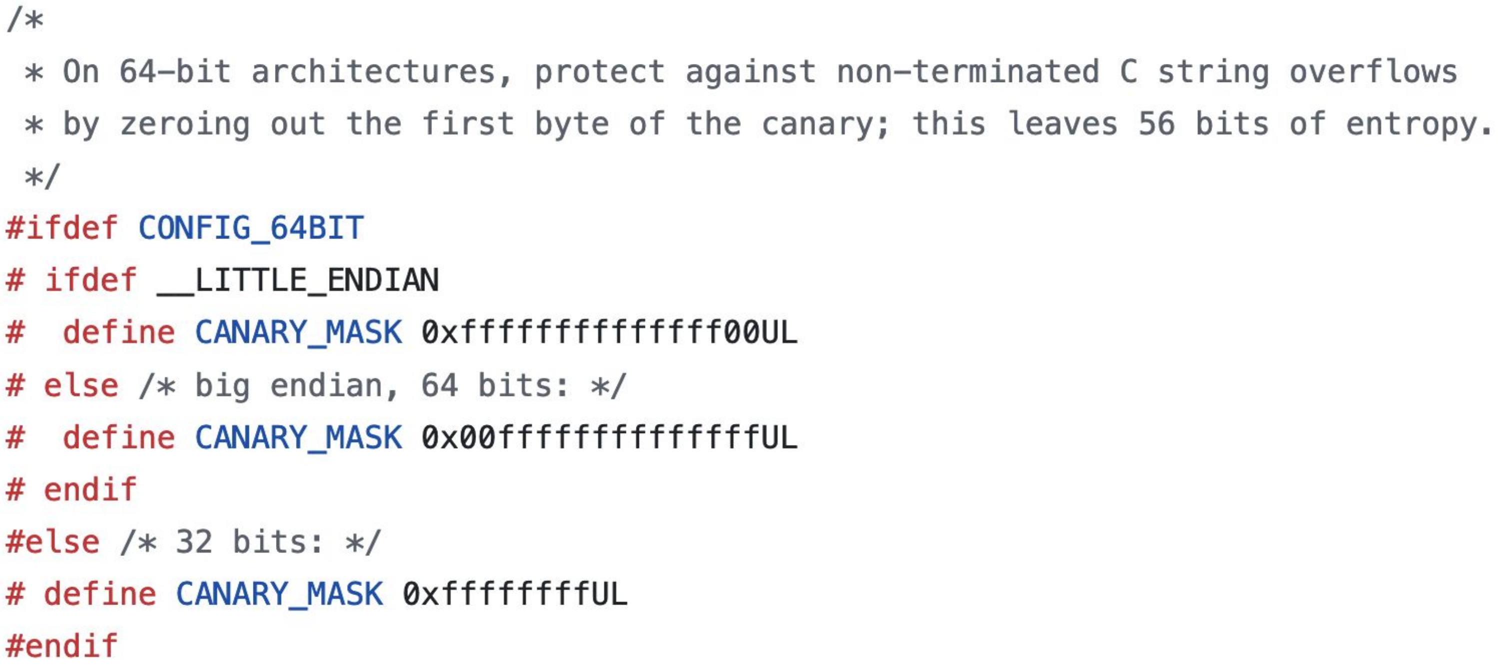
- **Random Canary** – The original concept for canary values took a **32**-bit pseudo random value generated by the **/dev/random** or **/dev/urandom** devices on a Linux operating system
- **Random XOR Canary** – The random canary concept was extended in StackGuard version 2 to provide slightly more protection by performing a **XOR** operation on the random canary value with the stored control data
- **Null Canary** – Canary value is set to **0x00000000** since most string functions terminate using a **null** value and should not overwrite the **return** address if the buffer must contain nulls
- **Terminator Canary** – Canary value is set to a combination of **null**, **CR**, **LF**, and **0xFF** and accounts for functions which do not simply terminate on nulls such as **gets()**

Terminator Canary

`0x000aff0d`

```
\x00: Terminates strcpy
\x0a: Terminates gets (LF)
\xff: Form feed
\x0d: Carriage return
```

Linux canary



```c
/*
 * On 64-bit architectures, protect against non-terminated C string overflows
 * by zeroing out the first byte of the canary; this leaves 56 bits of entropy.
 */
#ifdef CONFIG_64BIT
# ifdef __LITTLE_ENDIAN
#  define CANARY_MASK 0xffffffffffffff00UL
# else /* big endian, 64 bits: */
#  define CANARY_MASK 0x00ffffffffffffffUL
# endif
#else /* 32 bits: */
# define CANARY_MASK 0xffffffffUL
#endif
```

source: https://github.com/torvalds/linux/blob/master/include/linux/stackprotector.h#L23

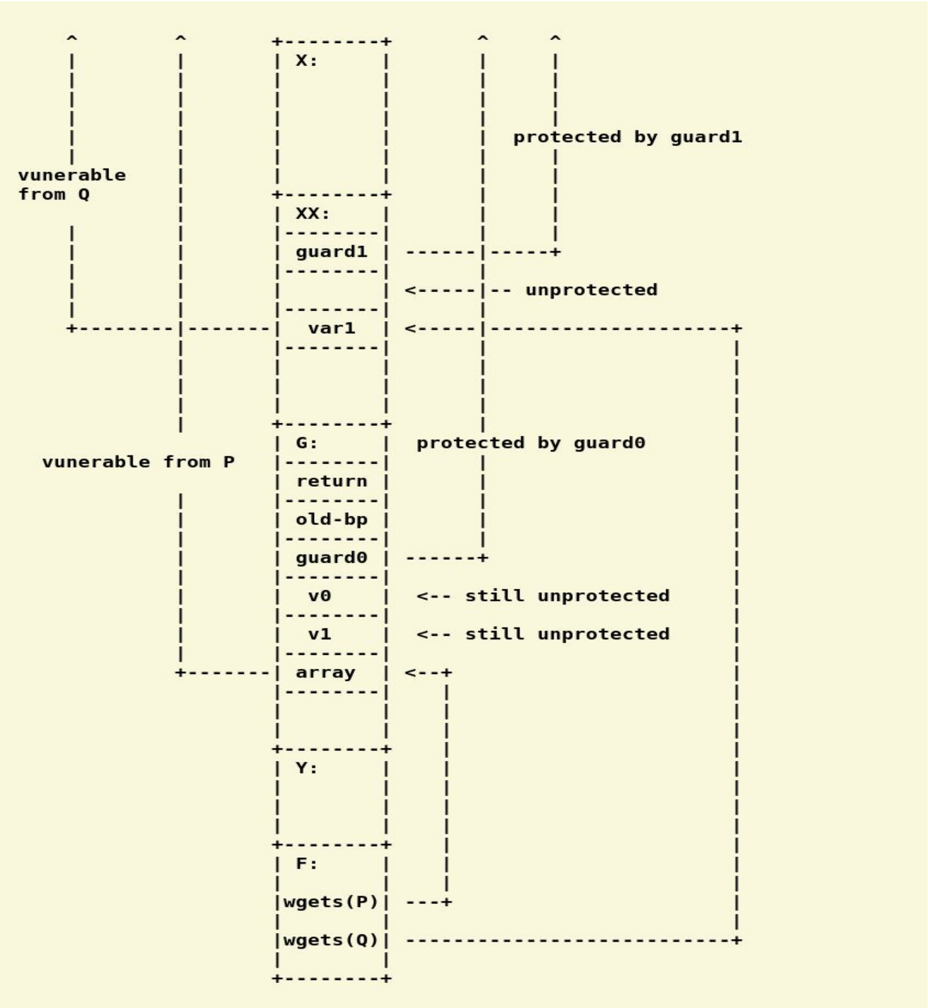
-fstack-protector-strong

- **`-fstack-protector`** is not enough
  - Adds stack protection to functions that have "`alloca`" or have a (signed or unsigned) char array with `size > 8 (SSP_BUFFER_SIZE)`
- **`-fstack-protector-all`** is an overkill
  - Adds stack protection to ALL functions.

-fstack-protector-strong

- **`-fstack-protector`** is not enough
  - Adds stack protection to functions that have "`alloca`" or have a (signed or unsigned) char array with `size > 8 (SSP_BUFFER_SIZE)`
- **`-fstack-protector-all`** is an overkill
  - Adds stack protection to ALL functions.

- **`-fstack-protector-strong`** was introduced by the Google Chrome OS team
  - Any function that declares any type or length of **local array**, even those in structs or unions
  - It will also protect functions that use a **local variable's address** in a function argument or on the right-hand side of an assignment
  - In addition, any function that uses **local register variables** will be protected

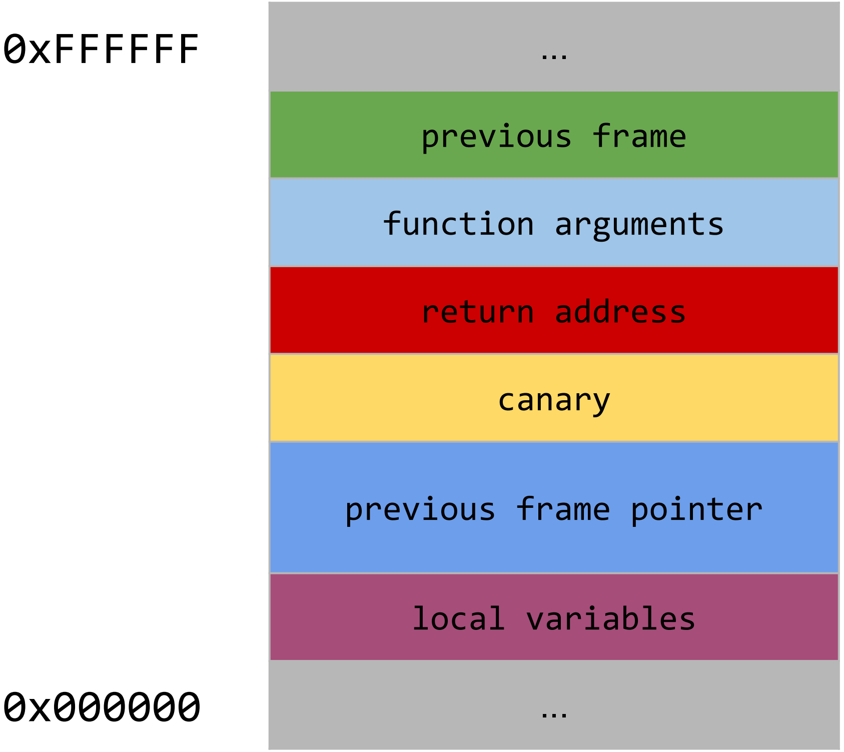


```
    ^        ^       +--------+       ^     ^
    |        |       | X:     |       |     |
    |        |       |        |       |     |
    |        |       |        |       |     |
    |        |       |        |       |     |
    |        |       |        |       |  protected by guard1
    |        |       |        |       |     |
vunerable    |       |        |       |     |
from Q       |       +--------+       |     |
             |       | XX:    |       |     |
    |        |       |--------|       |     |
    |        |       | guard1 | ------|-----+
    |        |       |--------|       |
    |        |       |        | <-----|-- unprotected
    |        |       |--------|       |
    +--------|-------|  var1  | <-----|--------------------+
             |       |--------|       |                    |
             |       |        |       |                    |
             |       |        |       |                    |
             |       |        |       |                    |
             |       +--------+       |                    |
                     | G:     |  protected by guard0       |
  vunerable from P   |--------|       |                    |
                     | return |       |                    |
             |       |--------|       |                    |
             |       | old-bp |       |                    |
             |       |--------|       |                    |
             |       | guard0 | ------+                    |
             |       |--------|                            |
             |       |  v0    |  <-- still unprotected     |
             |       |--------|                            |
             |       |  v1    |  <-- still unprotected     |
             |       |--------|                            |
             +-------| array  | <--+                       |
                     |--------|    |                       |
                     |        |    |                       |
                     |        |    |                       |
                     +--------+    |                       |
                     | Y:     |    |                       |
                     |        |    |                       |
                     |        |    |                       |
                     |        |    |                       |
                     +--------+    |                       |
                     | F:     |    |                       |
                     |        |    |                       |
                     |wgets(P)| ---+                       |
                     |        |                            |
                     |wgets(Q)| ----------------------------+
                     |        |
                     +--------+
```

source

Can we bypass stack canaries?

Frame Pointer Overwrite Attack



0xFFFFFF

…

previous frame

function arguments

return address

canary

previous frame pointer

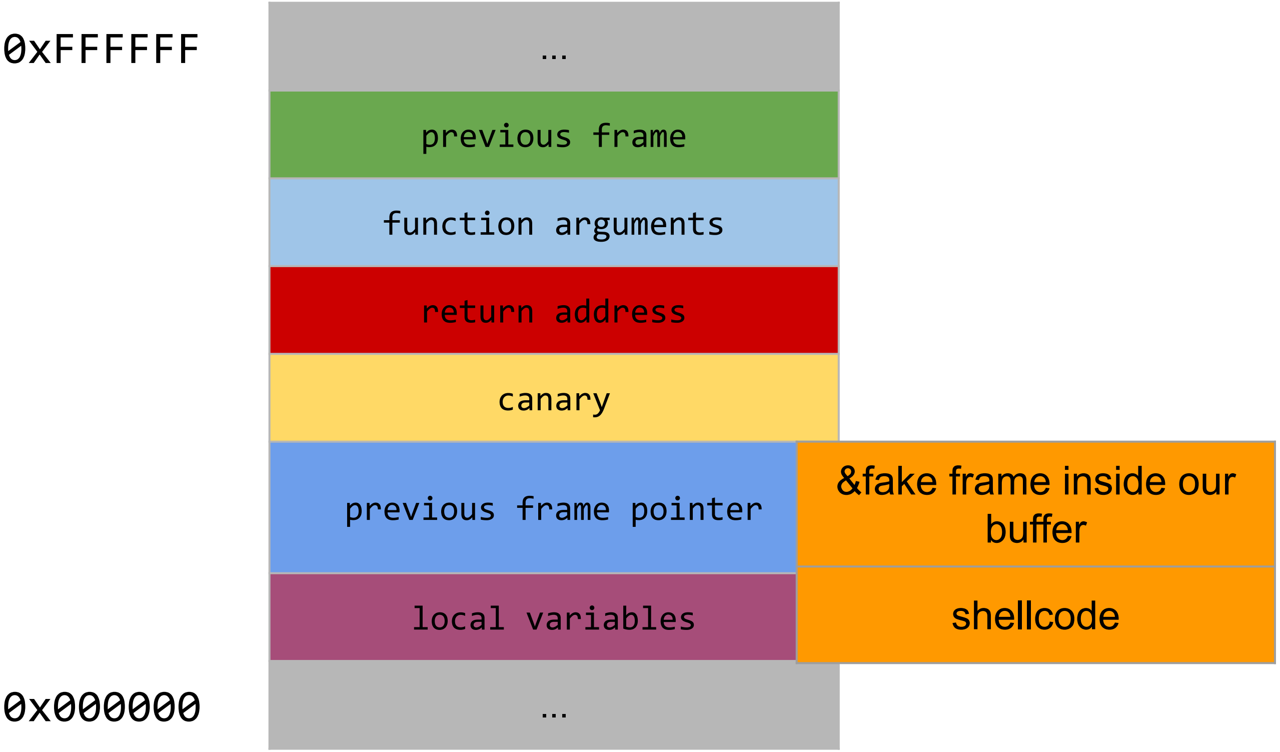
local variables

0x000000

…

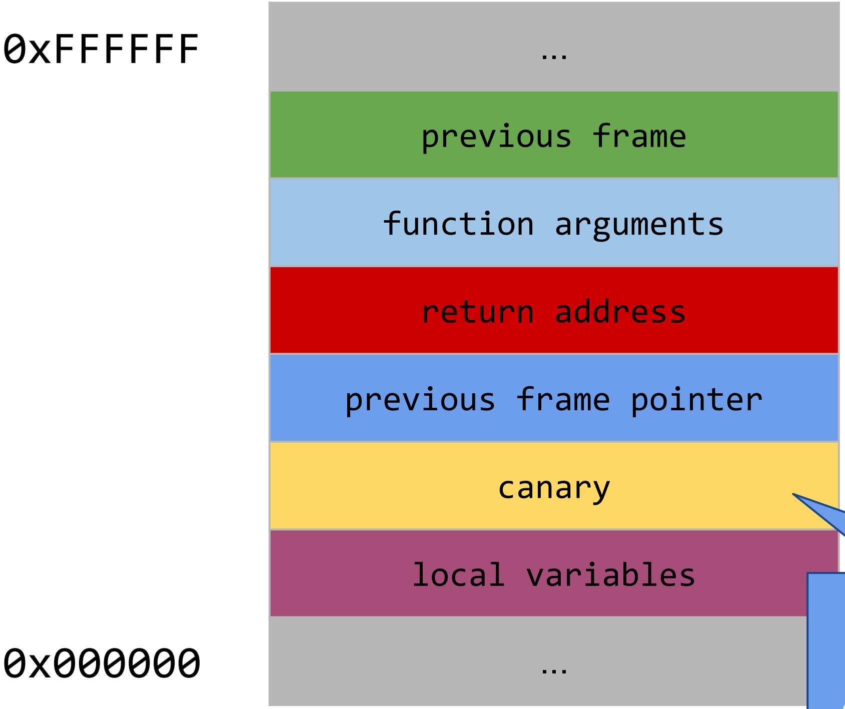
http://phrack.org/issues/55/8.html#article

Frame Pointer Overwrite Attack



http://phrack.org/issues/55/8.html#article

Frame Pointer Overwrite Attack



This was fixed to prevent overwriting frame pointers as well

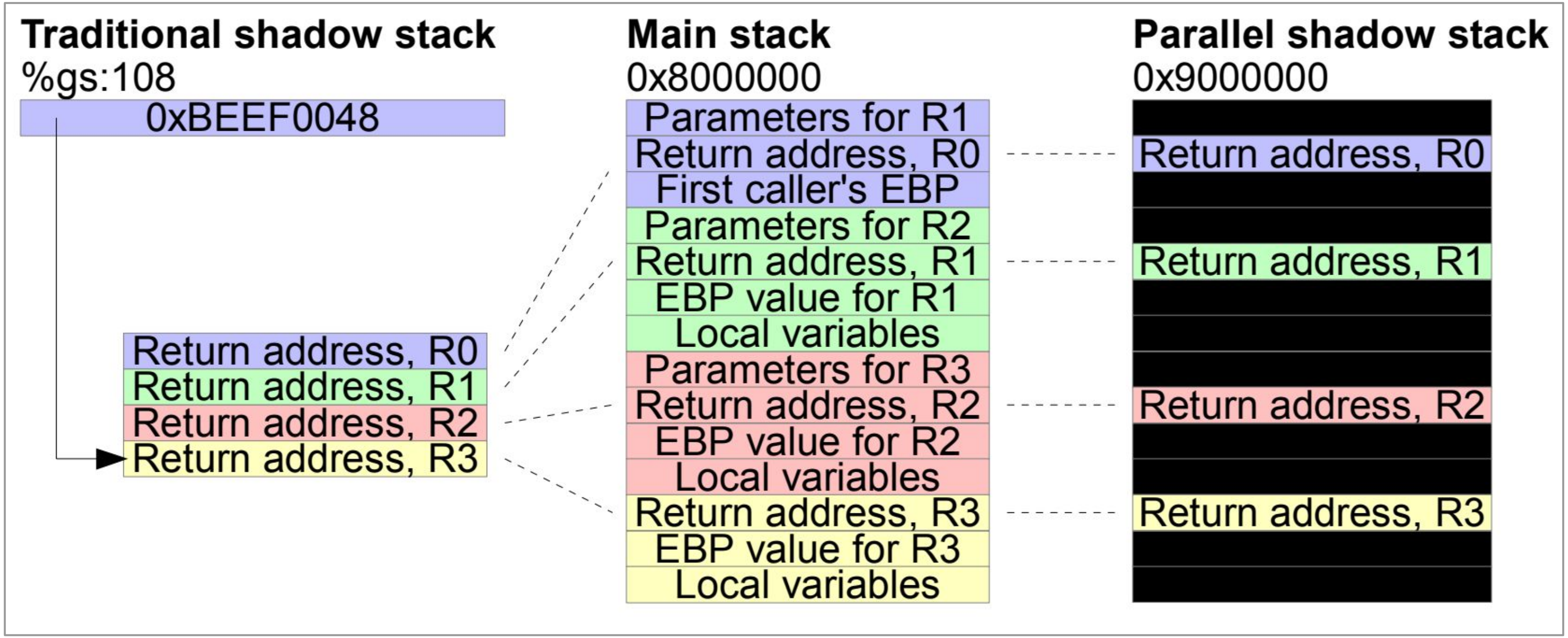
Other Pointers

- Global Offset Table (GOT)
  - Table of addresses which reside in the data section
  - helps with relocations in memory
- Function pointers
- Non-overflow exploits with arbitrary writes

http://phrack.org/issues/56/5.html#article

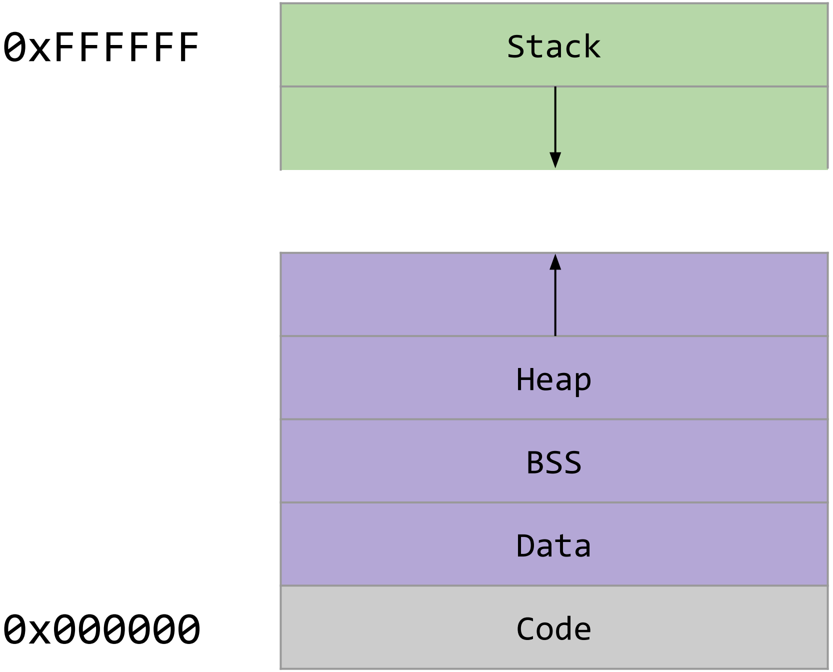
Shadow Stack

- Proposed a different defense in 2015 (but not implemented yet)

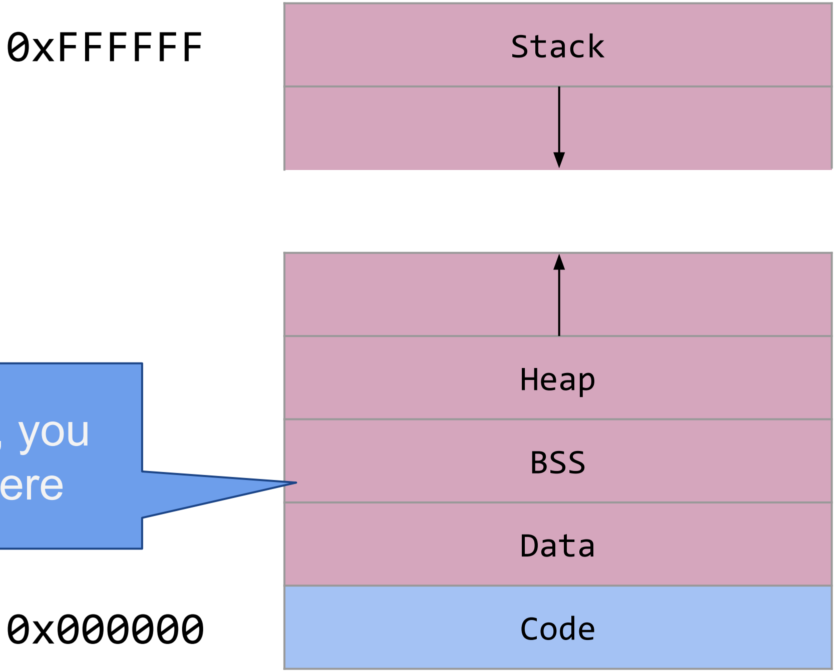


source: https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf

NOEXEC (W^X)



NOEXEC (W^X)



If you can write to it, you **cannot** execute there

`-z execstack`

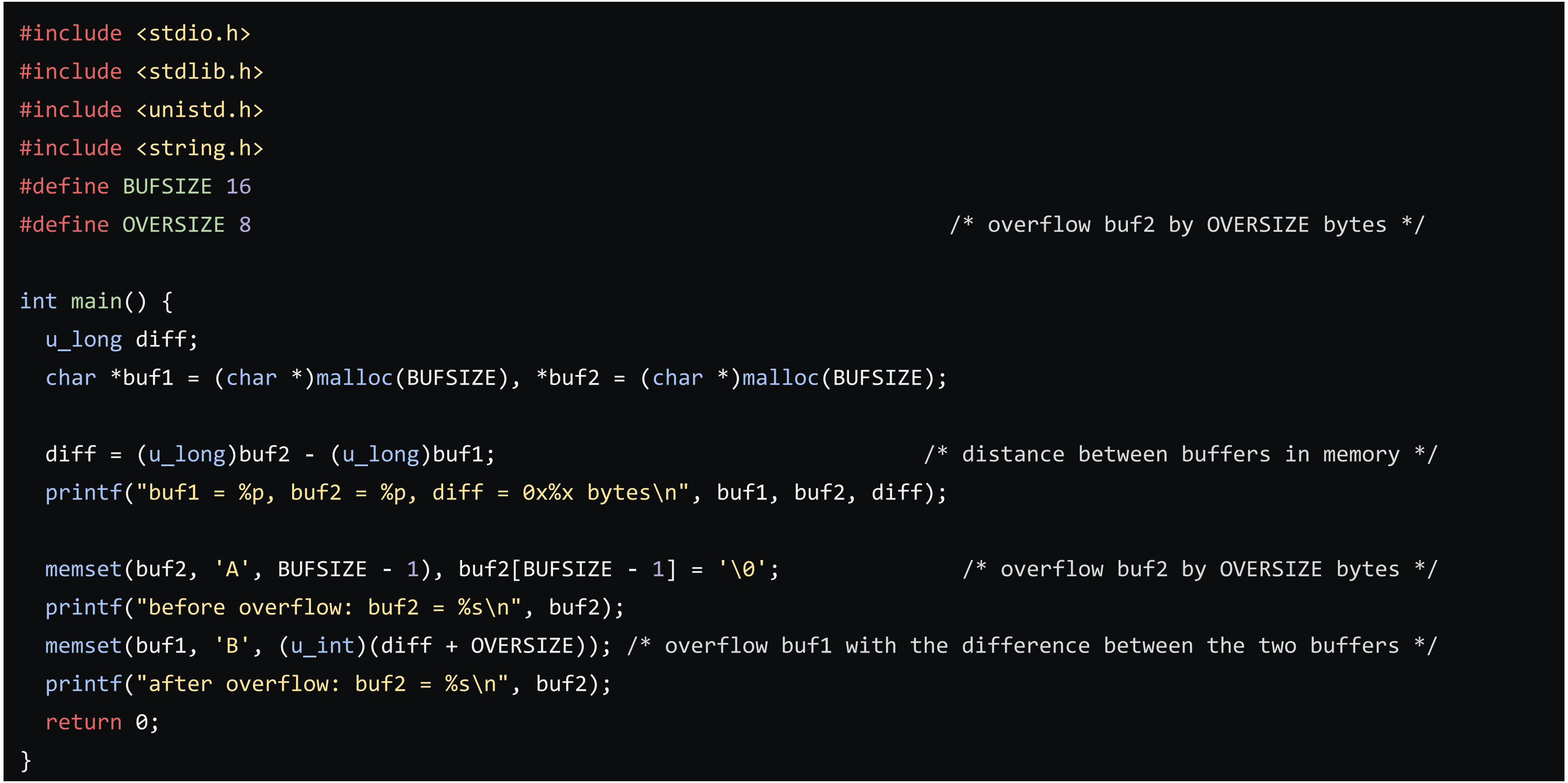
RW
RX

Address Space Layout Randomization (ASLR)

- Randomly arranges the address space positions of key data areas of a process
  - the base of the executable
  - the stack
  - the heap
  - libraries
- Discovering the address of your shellcode becomes a difficult task

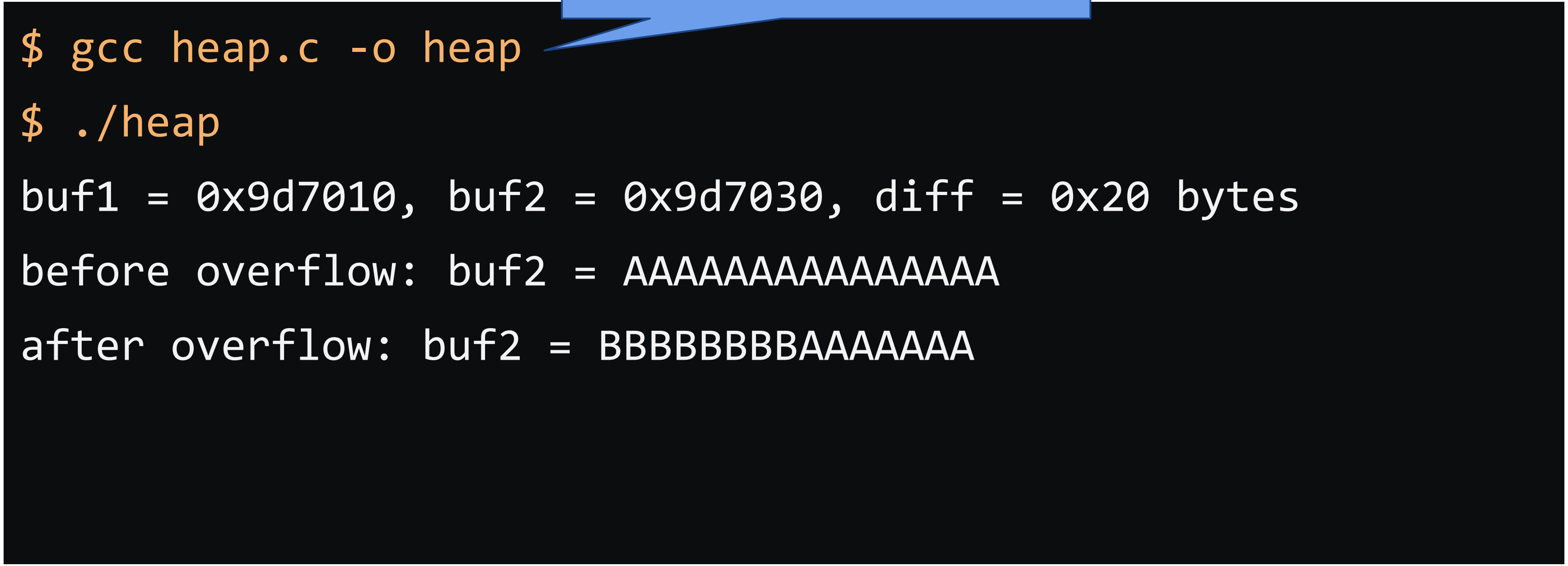
What about Heap-based Overflows?

Heap-based Overflows



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSIZE 16
#define OVERSIZE 8                                                      /* overflow buf2 by OVERSIZE bytes */

int main() {
  u_long diff;
  char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char *)malloc(BUFSIZE);

  diff = (u_long)buf2 - (u_long)buf1;                                 /* distance between buffers in memory */
  printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);

  memset(buf2, 'A', BUFSIZE - 1), buf2[BUFSIZE - 1] = '\0';              /* overflow buf2 by OVERSIZE bytes */
  printf("before overflow: buf2 = %s\n", buf2);
  memset(buf1, 'B', (u_int)(diff + OVERSIZE)); /* overflow buf1 with the difference between the two buffers */
  printf("after overflow: buf2 = %s\n", buf2);
  return 0;
}
```

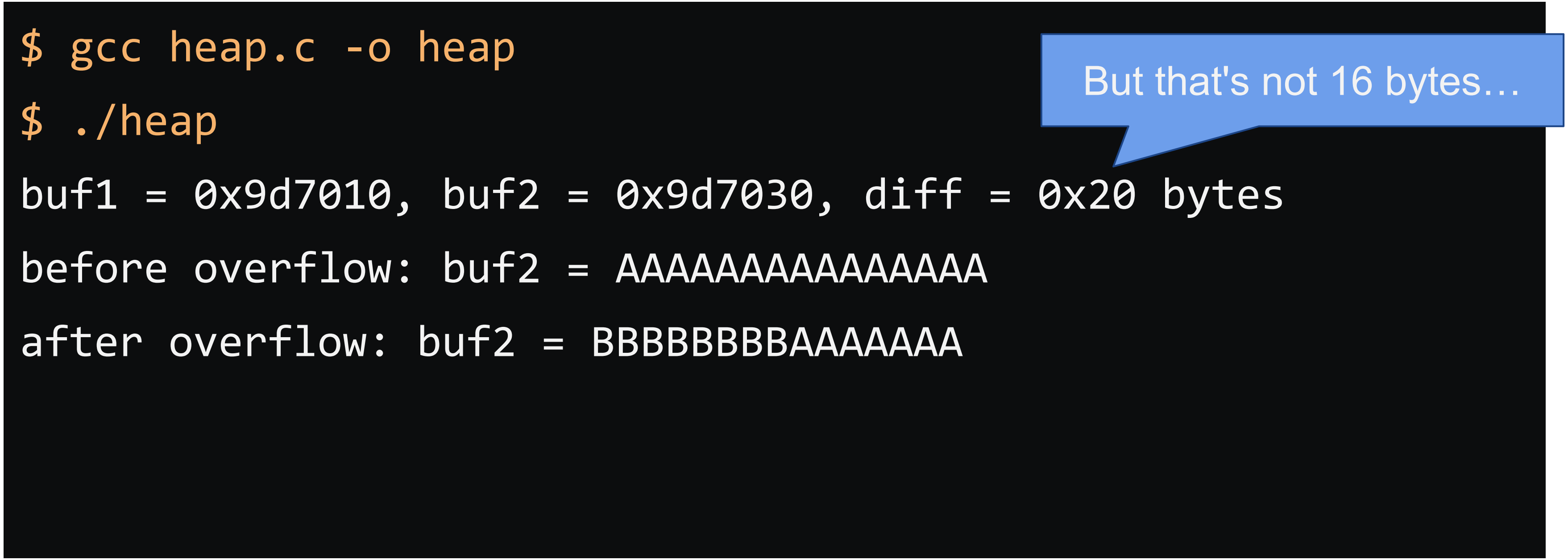
Overflow into another buffer



```
$ gcc heap.c -o heap
$ ./heap
buf1 = 0x9d7010, buf2 = 0x9d7030, diff = 0x20 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow: buf2 = BBBBBBBBAAAAAAA
```

No flag for gcc protections!

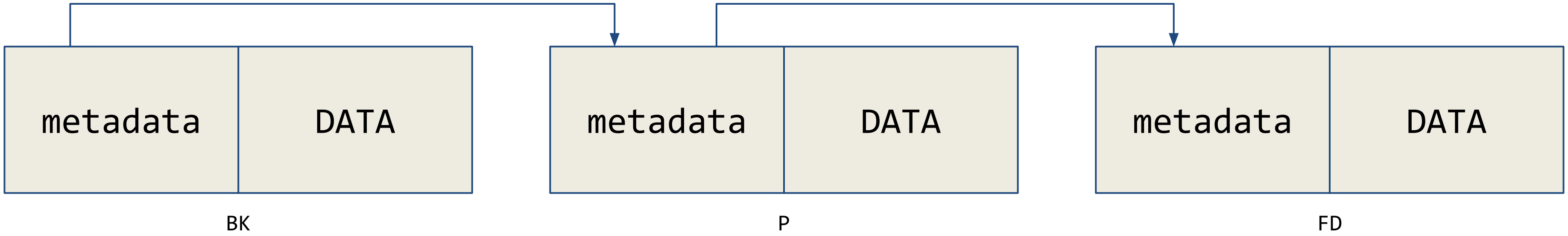
Overflow into another buffer



```
$ gcc heap.c -o heap
$ ./heap
buf1 = 0x9d7010, buf2 = 0x9d7030, diff = 0x20 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow: buf2 = BBBBBBBAAAAAAA
```

But that's not 16 bytes…

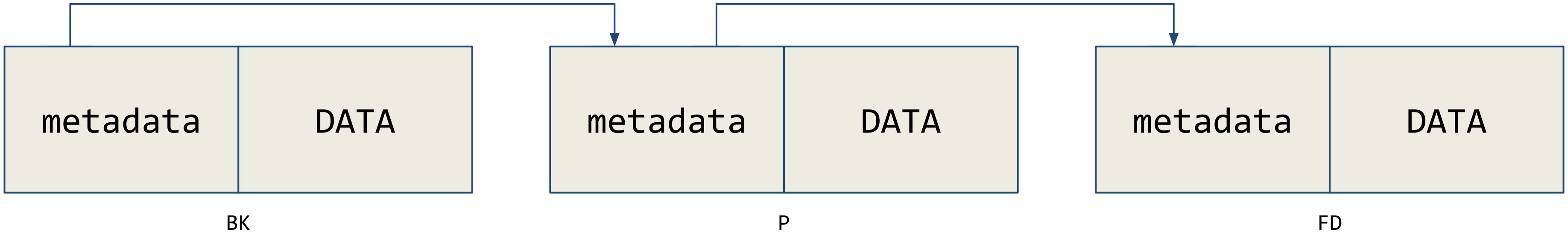
How does malloc/free work?



free()

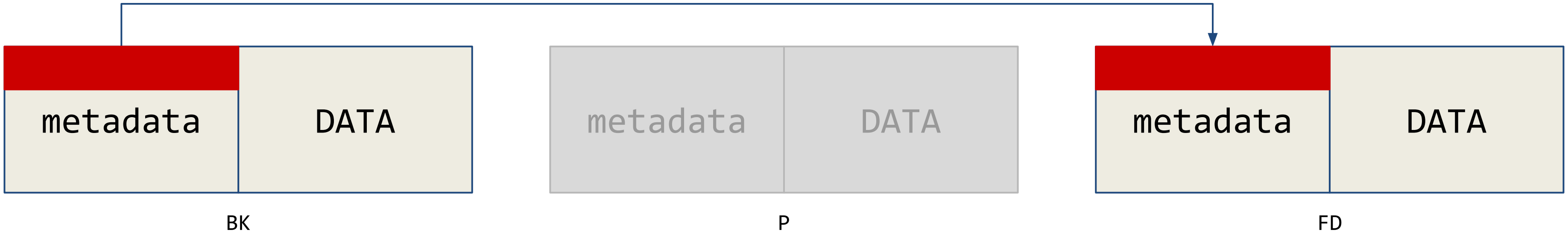
```
#define unlink( P, BK, FD ) {
    [1] BK = P->bk;
    [2] FD = P->fd;
    [3] FD->bk = BK;
    [4] BK->fd = FD;
}
```

Similar to CSC 216, unlink will remove P from the linked list and connect BK to FD



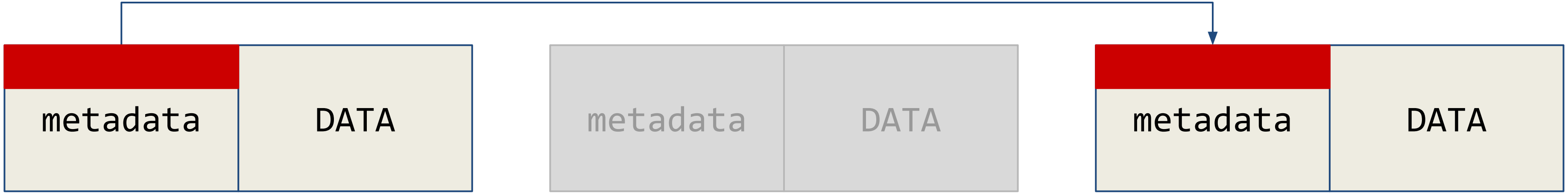
free()

```
#define unlink( P, BK, FD ) {
    [1] BK = P->bk;
    [2] FD = P->fd;
    [3] FD->bk = BK;
    [4] BK->fd = FD;
}
```



free()

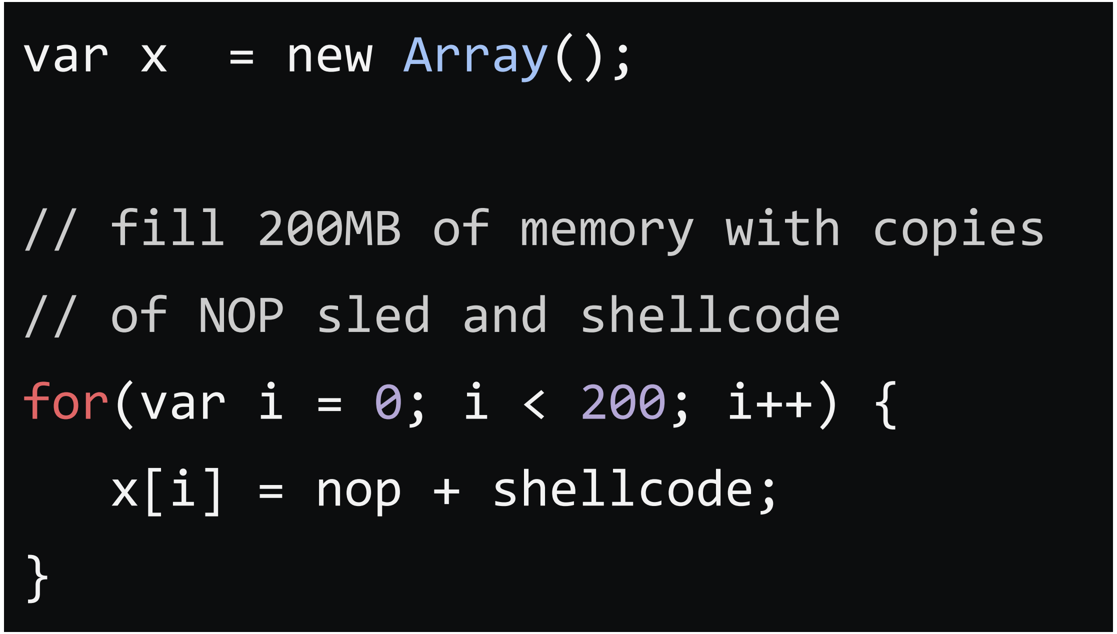
```
#define unlink( P, BK, FD ) {
    [1] BK = P->bk;
    [2] FD = P->fd;
    [3] FD->bk = BK;
    [4] BK->fd = FD;
}
```



Arbitrary write!!!

Let’s break ASLR in the heap!

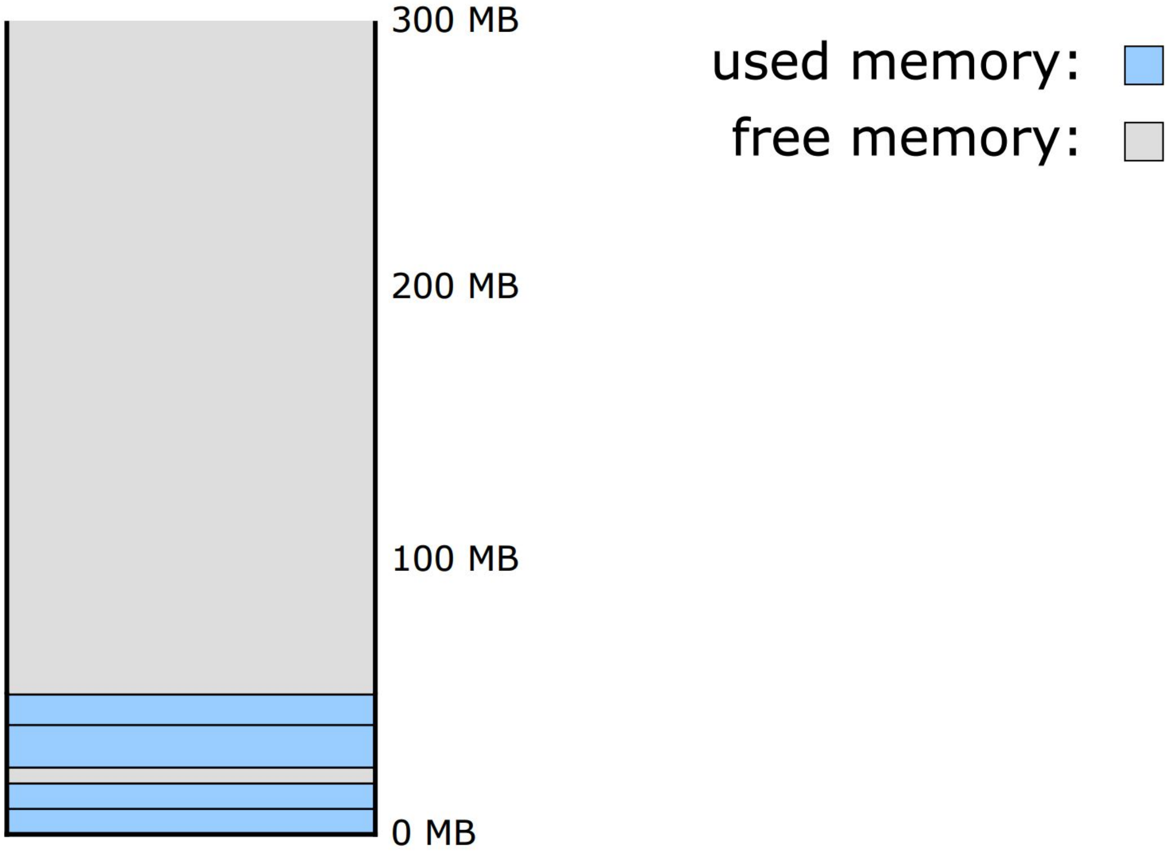
Heap Spraying



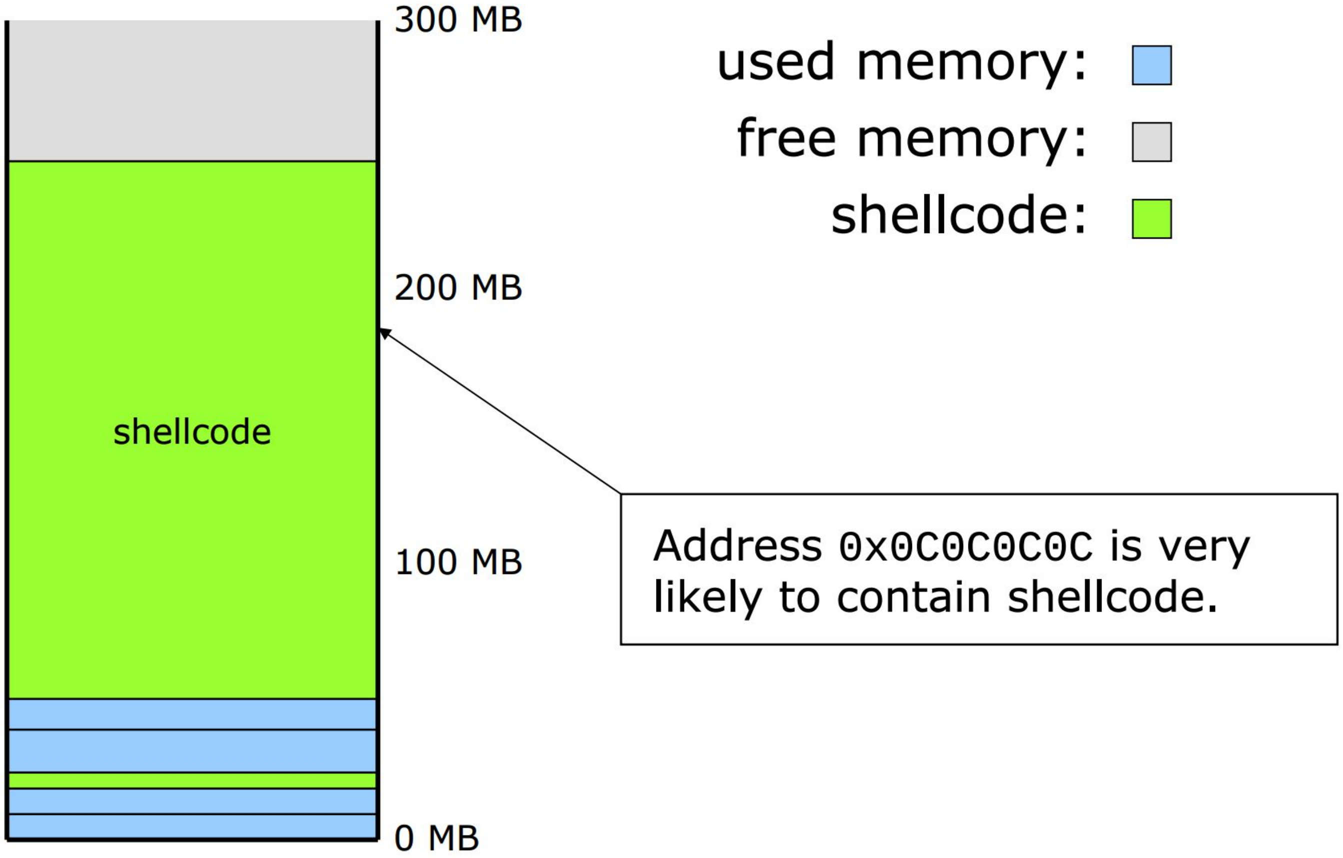
```
var x  = new Array();

// fill 200MB of memory with copies
// of NOP sled and shellcode
for(var i = 0; i < 200; i++) {
   x[i] = nop + shellcode;
}
```

source: [Heap Feng Shui in Javascript]

Heap Spraying - Normal Heap



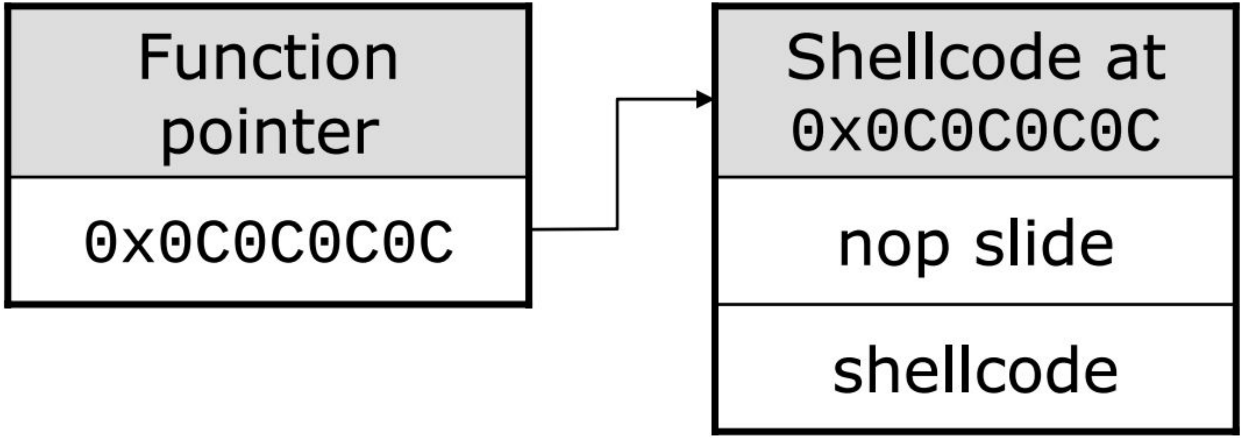
Heap Spraying - Heap Sprayed



Address 0x0C0C0C0C is very likely to contain shellcode.

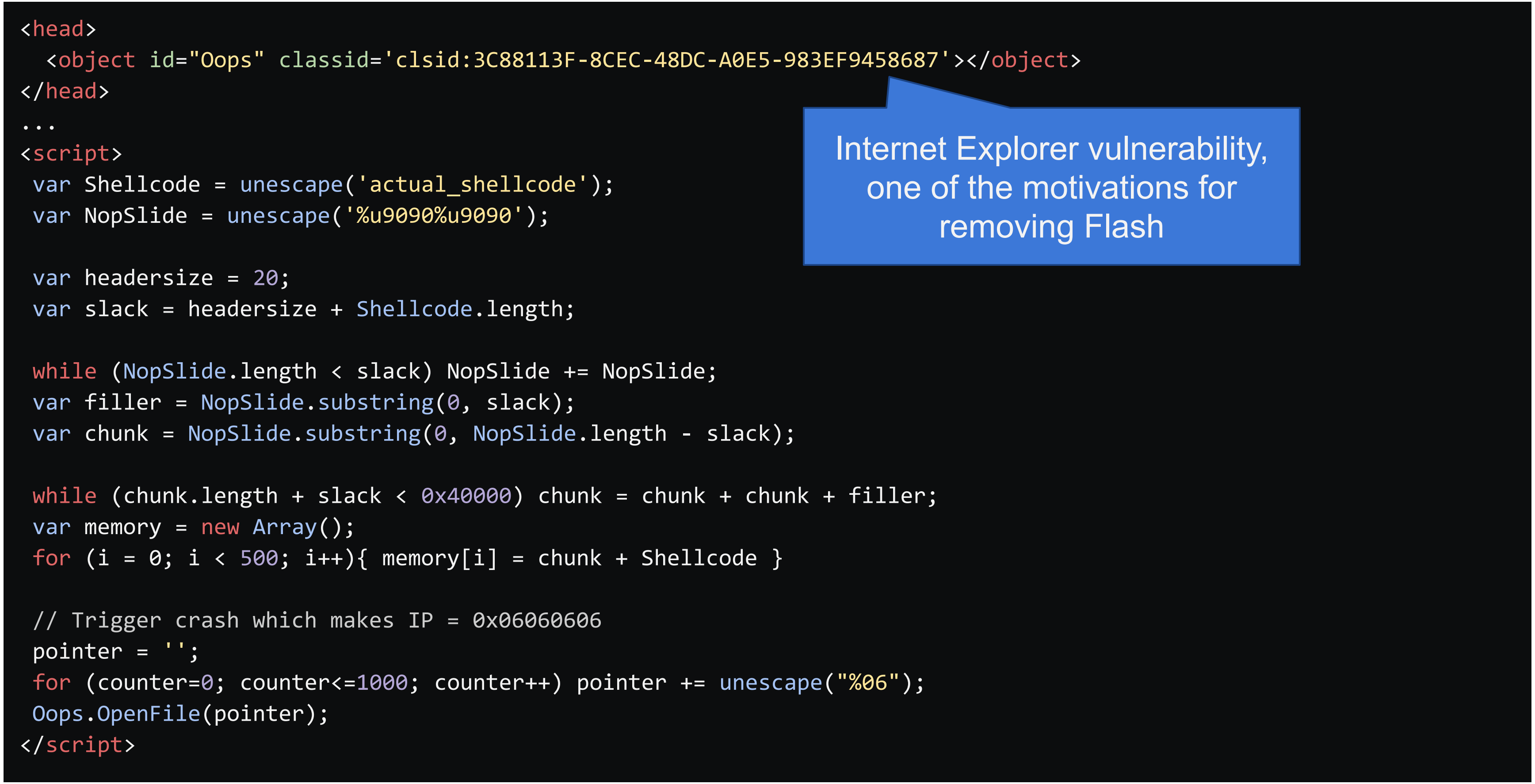
Heap Spraying Strategy

1. "Spray" the heap with 200MB of **`nopsled + shellcode`**
2. Overwrite a function pointer with **`0x0c0c0c0c`**
3. Arrange for the pointer to be called



source: Putting Needles in the Haystack (link)

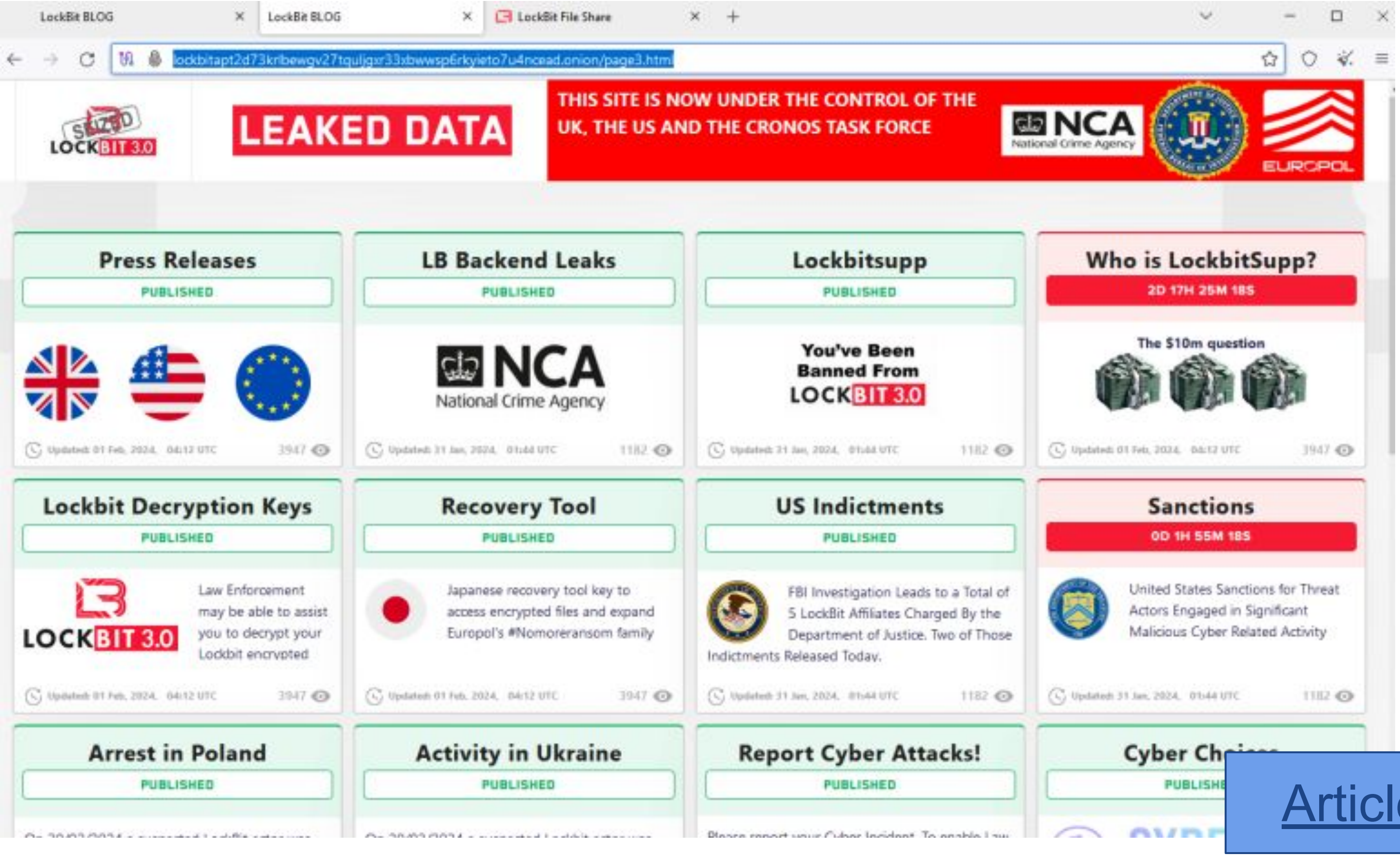
ActiveX Heap Spray



```
<head>
  <object id="Oops" classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687'></object>
</head>
...
<script>
 var Shellcode = unescape('actual_shellcode');
 var NopSlide = unescape('%u9090%u9090');

 var headersize = 20;
 var slack = headersize + Shellcode.length;

 while (NopSlide.length < slack) NopSlide += NopSlide;
 var filler = NopSlide.substring(0, slack);
 var chunk = NopSlide.substring(0, NopSlide.length - slack);

 while (chunk.length + slack < 0x40000) chunk = chunk + chunk + filler;
 var memory = new Array();
 for (i = 0; i < 500; i++){ memory[i] = chunk + Shellcode }

 // Trigger crash which makes IP = 0x06060606
 pointer = '';
 for (counter=0; counter<=1000; counter++) pointer += unescape("%06");
 Oops.OpenFile(pointer);
</script>
```

Internet Explorer vulnerability, one of the motivations for removing Flash

Security Zen - Feds Seize LockBit Websites, Offer Decryption Tools



LEAKED DATA

THIS SITE IS NOW UNDER THE CONTROL OF THE UK, THE US AND THE CRONOS TASK FORCE

Press Releases
PUBLISHED
Updated: 01 Feb, 2024, 04:12 UTC 3947

LB Backend Leaks
PUBLISHED
NCA National Crime Agency
Updated: 31 Jan, 2024, 01:44 UTC 1182

Lockbitsupp
PUBLISHED
You've Been Banned From LOCKBIT 3.0
Updated: 31 Jan, 2024, 01:44 UTC 1182

Who is LockbitSupp?
2D 17H 25M 18S
The $10m question
Updated: 01 Feb, 2024, 04:12 UTC 3947

Lockbit Decryption Keys
PUBLISHED
LOCKBIT 3.0
Law Enforcement may be able to assist you to decrypt your Lockbit encrypted
Updated: 01 Feb, 2024, 04:12 UTC 3947

Recovery Tool
PUBLISHED
Japanese recovery tool key to access encrypted files and expand Europol's #Nomoreransom family
Updated: 01 Feb, 2024, 04:12 UTC 3947

US Indictments
PUBLISHED
FBI Investigation Leads to a Total of 5 LockBit Affiliates Charged By the Department of Justice. Two of Those Indictments Released Today.
Updated: 31 Jan, 2024, 01:44 UTC 1182

Sanctions
0D 1H 55M 18S
United States Sanctions for Threat Actors Engaged in Significant Malicious Cyber Related Activity
Updated: 31 Jan, 2024, 01:44 UTC 1182

Arrest in Poland
PUBLISHED

Activity in Ukraine
PUBLISHED

Report Cyber Attacks!
PUBLISHED

Cyber Ch
PUBLISHE

[Article Link]